

EE155 / COMP122: Parallel Computing

Lab 3: Matrix multiplication

In this lab, you will experiment with multiplying matrices. The files `matrix.cxx` and `matrix.hxx` (which you should not touch) provide a class *Matrix*. It provides various basic capabilities:

- Instantiate a matrix. You can supply two arguments: the number of rows and columns. In this lab and the next two, all of our matrices are square. You can thus simply instantiate, e.g., `Matrix(128)` to create a 128x128 matrix.
- Various initialization routines. `Matrix::init_identity()` initializes it to the identity matrix; `::init_random()` fills it with random real numbers in $[0,1]$; and `::init_cyclic_order()` fills the first row with $[0,1,2,\dots]$, the second row with $[1,2,3,\dots]$, the third row with $[3,4,5,\dots]$, etc.
- `operator()`. Internally, the matrix stores its data as one long 1D vector, rather than as a 2D array. However, we give you `operator()` to access it easily with two indices. E.g., you can do `foo=my_matrix(3,5)`, or `my_matrix(3,5)=2`. I used `operator()` rather than `operator[]` since the latter is only allowed to accept one parameter.
- `row_str(int row)` returns a printable string for the given row; `str()` returns a printable string for the entire matrix.
- It notably does *not* provide the routines `Matrix::mpy1()` and `Matrix::mpy2()`, which are for you to write.

We also provide you the file `matrix_mpy.cxx`, which you should not touch. It supplies

- The function `main()`. It creates the matrices for you to multiply and runs a slow-but-reliable method to create a golden-reference output. Then it runs both of the methods you write, `Matrix::mpy1()` and `Matrix::mpy2()`, to check that they work and to time them.
- `Matrix::mpy_dumb()`, as noted above, implements a very simple matrix multiplication scheme that is single threaded and not blocked; it is simple and reliable but slow.
- `Run_mpy2()` calls your `Matrix::mpy2()` for various matrix sizes and numbers of threads.

Finally, the code that you write should go into `matrix_mpy_user.cxx`. You should implement:

- `Matrix::mpy1()`: a single-threaded blocked matrix multiply that uses the ordering rB, kB, cB, r, k, c.
- `Matrix::mpy2()`: a multithreaded implementation that spawns as many threads as requested.
- Remember, as we discussed in class, that you must zero the matrix before your calculations.

The program `matrix_mpy.cxx` will then collect the following timing data:

- The `mpy_dumb()` algorithm for matrices of size 1Kx1K
- `mpy_dumb()` and both of your algorithms for a matrix of size 2Kx2K. It will use a block size of 128x128; and run `mpy2()` using 1, 2, 4, 8 and 16 threads.

Reference results (for answering all questions below):

	1Kx1K	2Kx2K
<code>mpy_dumb</code>	1.7 sec	68 sec
<code>mpy1 (1T, blocked)</code>	N/A	9.4 sec

	1 thread	2 threads	4 threads	8 threads	16 threads
mpy2, 2Kx2K	9.1 sec	4.6 sec	2.4 sec	2.4 sec	2.4 sec

Background. Each lab machine has one chip with four dual-threaded cores. Each core has its own 32KB L1D and 256KB L2, and there is an additional 8MB of L3 cache shared among the four cores. Also note that for the questions below, we are *not* expecting you to take the time to analyze inner loops in assembly code and count instructions and issuing; rough answers are thus good enough.

Grading. Getting the code to work is worth 45 points (please submit your final timings). You then get 15 points for having chosen, and explaining, how you divided the work among threads in a reasonable way. Finally, using the reference results above, please answer the following questions for the remaining 40 points:

Question #1. *Mpy_dumb()* got roughly 40x slower when moving from 1Kx1K matrices to 2Kx2K matrices. Let's try to figure out why.

- How many times more floating-point operations did we have? (5 points)
- Consider the actual data-access bandwidth to the *B* matrix (i.e., the bandwidth we get, given the cache or memory that we mostly access *B* from). How much does this BW change when we move from 1Kx1K matrices to 2Kx2K matrices? Assume a clock speed of 4GHz (5 points).
- How much of the 40x slowdown have we now explained if we just multiply the two slowdown numbers above? Is there any reason why a simple multiplication may not be the correct way to combine the two numbers (hint: think about whether programs are compute limited, memory limited or both)? (5 points)

Question #2. (10 points) On the 2Kx2K matrix, *mpy1()* ran almost 7x faster than *mpy_dumb()*. How might you qualitatively explain such a big difference? (You need not explain why it's precisely 7x)?

Question #3. Many people saw a near-linear speedup going from 1 to 2 to 4 threads with *mpy2()*, and then saw essentially no speedup from going to 8 or more threads.

- Explain the near-linear speedup going from 1 to 2 to 4 threads. Feel free to reference our conclusion from the class discussion about the bottlenecks of matrix multiplication (5 points).
- Explain the lack of speedup going to 8 or more threads vs. 4T. Remember that our CPU supports two threads/core with hyperthreading (5 points).
- How might the use of AVX instructions (Intel's SIMD instruction set introduced in 2011) affect this data? Specifically, might we expect performance to saturate at fewer threads? Answering qualitatively is good enough (5 points).

Logistics:

- The lab assignment is due at midnight on the day indicated by the class calendar. You must work on this assignment on your own.
- Remember to compile with the line `c++ -pthread -std=c++20 -O2 matrix_mpy.cxx matrix.cxx matrix_mpy_user.cxx ee155_utils.cxx`. You may add debugging flags as needed (but remove them when you run your final benchmarks).

- Submit your `matrix_mpy_user.cxx` via **provide**. You should also submit a copy of your report as a PDF.
- As usual, use any Linux box in the lab room. Before you collect benchmarking data, you should reboot the machine.